Multiclass classification on CIFAR-100 dataset using back-propagation

Rishabh Bhattacharya
Department of Mechanical and Aerospace Engineering
University of California, San Diego
La Jolla, CA 92092
`ribhattacharya@ucsd.edu`

Shrey Kansal
Department of Mechanical and Aerospace Engineering
University of California, San Diego
La Jolla, CA 92092
`skansal@ucsd.edu`

Abstract

A multi-layer Neural Network is implemented to classify the CIFAR-100 dataset into 20 and 100 classes. Hyperparameter tuning is carried out sequentially to improve the accuracy of the network and compare the effects of different hyperparameters for classifying the dataset into 20 and 100 classes. Finally, the network topology is altered to study its effects, and the results are reported. We have achieved about 25% to 29% accuracy for the 20-class classification with different hyperparameter combinations and about 20% with 100 classes.

1 Introduction

Given the CIFAR dataset consisting of 50000 training and 10000 test examples respectively, a 2-layer Neural Network was trained to classify the data into 20 classes. Multiple hyperparameters including learning rate, momentum, L2 regularization penalty, number of hidden layers and units were sequentially tuned to gradually improve the network's accuracy. Early stopping was implemented to save the best model for test data. Gradient descent with momentum was implemented to minimize the oscillations of gradient vectors and to ensure faster convergence of the solution. Adding a regularization term helped to prevent overfitting by limiting the size of the weights in simultaneous iterations.

The weights of the network were updated using backpropagation to minimize the error between predicted probabilities and given target labels. Essentially, a multi-layer Neural Network is composed of numerous connections between adjacent layers, which are in-turn the weights of the network. The backpropagation algorithm works of the backbone of chain-rule of differential calculus. The loss function is differentiated with respect to each weight in the network. These derivatives are then used to update the weights through a gradient descent algorithm, which moves the weights in the desired direction. In order for the network to predict correct labels, multiple forward and backward passes are made to update the weights.

The effect of learning rate (α) and momentum (γ) is studied first, and accuracy of about 25% is achieved with best possible hyperparameters. The validation set accuracy is further increased to under 26% by adding regularization. Changing the activation function of hidden layer from `tanh` to `ReLU` had the most prominent effect of increasing the accuracy to about 29%. While, changing the network topology had little effect on the accuracy, training with 100 classes reduced the accuracy to about 20%.

2 Related Work

We have referered to Prof. Gary Cotrell's lecture notes on Lec 3: Backpropagation and Lec 4A: Improving generalization.

We implemented backpropagation and momentum using Lec 3, and used it to compute the gradients of the concerned quantities during the weight update.

We incorporated regularization using Lec 4A, and used the text to help us tune our hyperparameters.

3 Dataset

We use the CIFAR-100 dataset consisting of images, belonging to 100 fine-grained classes, further grouped into 20 coarse-grained classes. The dataset consists of 600 images each from the 100 classes. The images have 3 color channels (R, G, B) and are of 32 $\times$ 32 size. It is a challenging dataset for training and testing neural networks due to limited amount of data for each class and small image size. We dedicate 10000 samples to the test set and the remaining 50000 images are further randomly grouped into training (80%) and validation (20%) set.

Z-score normalization is implemented on a per-image, per-channel basis for the entire dataset. This normalization ensures 0 mean and a standard deviation of about 1. In our dataset, each channel of each image is normalized separately as each channel of any image will have a distinct distribution. This technique would ensure that each feature in the image is treated equally.

A random image was selected to report the mean and standard deviation of each channel in Table 1.

Table 1: Dataset- mean and standard deviation

Channel	**Mean**	**Standard Deviation**
Red	0.0	1.0
Green	0.0	1.0
Blue	0.0	1.0

4 Gradient check

In this section, we compare our weights obtained by back-propagation and via numerical approximation. We manually change one weight in the whole model, to check if the difference between our values is within the order $\mathcal{O}(\epsilon^2)$, where $\epsilon = 0.01$.

Table 2: Gradient comparison (Numerical vs back-propagation)

Weight type	**Numerical gradient**	**Backprop gradient**	**Absolute difference**
Output bias	-0.050007452	-0.050007740	0.000000288
Hidden bias	-0.001354726	-0.001354793	0.000000066
Hidden to output	0.001833257	0.001833293	0.000000036
Hidden to output	-0.001040988	-0.001041008	0.000000020
Input to hidden	-0.000608575	-0.000608623	0.000000048
Input to hidden	-0.000534352	-0.000534341	0.000000011

We see from Table 2 that all the absolute differences are less than 0.0001 (ϵ^2). Thus we can stay confident that our gradients are being calculated properly via back-propagation.

5 Testing momentum

Momentum γ is applied to speed up the convergence of the gradient descent algorithm and preventing the solution to be stuck in a local minimum or making oscillations in the optimization contour. Sometimes, the momentum term overshoots the solution to a local minimum, and it should be applied conservatively. For the given dataset, the effect of momentum were studied by varying it along with different learning rates.

5.1 Hyperparameter tuning

We tuned the momentum coefficient γ simultaneously with the learning rate to maintain convergence of gradient descent before 100 epochs. While $\gamma < 1$ to prevent overshooting, lowering it beyond a point delays the convergence of the solution; albeit, with minimal improvement in accuracy. Hence, we restricted $\gamma \in \{0.7, 0.8, 0.9\}$.

Table 3: Hyperparameter tuning for momentum

Momentum γ (λ)	α	**Validation accuracy**	**Early stop epoch**
0.7	0.001	25.69%	-
0.7	0.005	25.30%	21
0.8	**0.001**	**25.85%**	**70**
0.8	0.005	26.32%	14
0.9	0.001	25.56%	35
0.9	0.005	24.77%	07

As apparent from Table 3, we selected $\gamma = 0.8$ and $\alpha = 0.001$ as our best parameters for this part. While decreasing γ from 0.9 to 0.8 did increase the accuracy on the validation set, lowering it further 0.7 nullified the effect of momentum and gradient descent seems to have settled at a local minimum. Furthermore, it is seen that a lower α delays the convergence and improves the prediction accuracy as well.

5.2 Results

Based on the best parameters ($\gamma = 0.8$ and $\alpha = 0.001$), accuracy and loss plots for both, training and validation sets are obtained (Fig 1 and Fig 2).

It can be seen from the plots that, the loss is minimized around epoch 70 and the accuracy peaks around that epoch as well.

The test set accuracy at the selected parameters was found to be 25.41%.

6 Regularization

Regularization is used to prevent overfitting, by adding a penalty term to the loss function. This penalty is generally added as a L_1 ($|W|$) or L_2 ($||W||_2^2$) norm of the weights.

A high penalty penalizes the weights more, preventing it to learn and resulting in a very sinple model (underfitting). A low penalty allows the model to overfit the training data, since weights become excessively large for a select few features. This leads to poor generalization on unseen data. Thus it is critical to tune the *right* penalty value. For this implementation, we have used the panalty value for our gradient calculation, but have skipped it for the loss computation.

6.1 Hyperparameter tuning

We observe from Table 4 that for lower learning rates, early stop happens at much later epochs. Higher L_2 penalties push the early stop farther. In fact, for $\lambda = 0.01$ and $\alpha = 0.001$, we do not

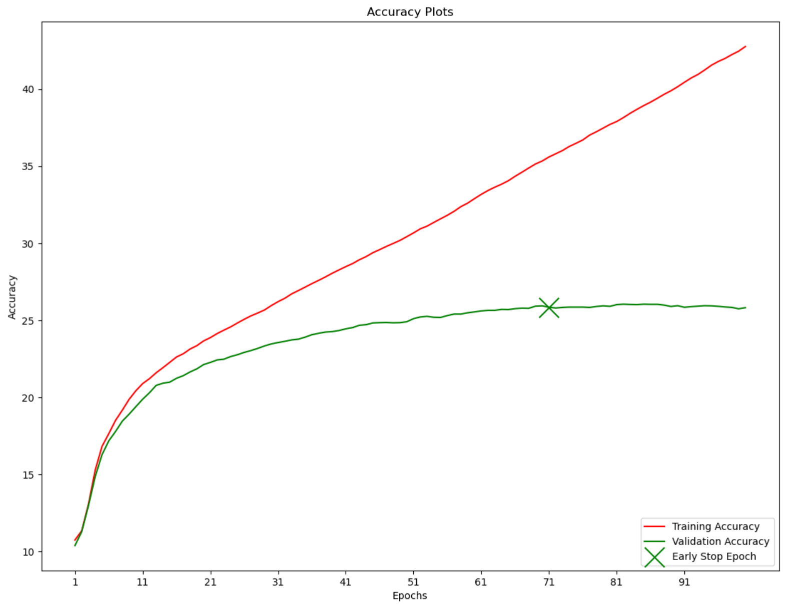

Figure 1: Training and Validation Accuracy for Momentum Experiment

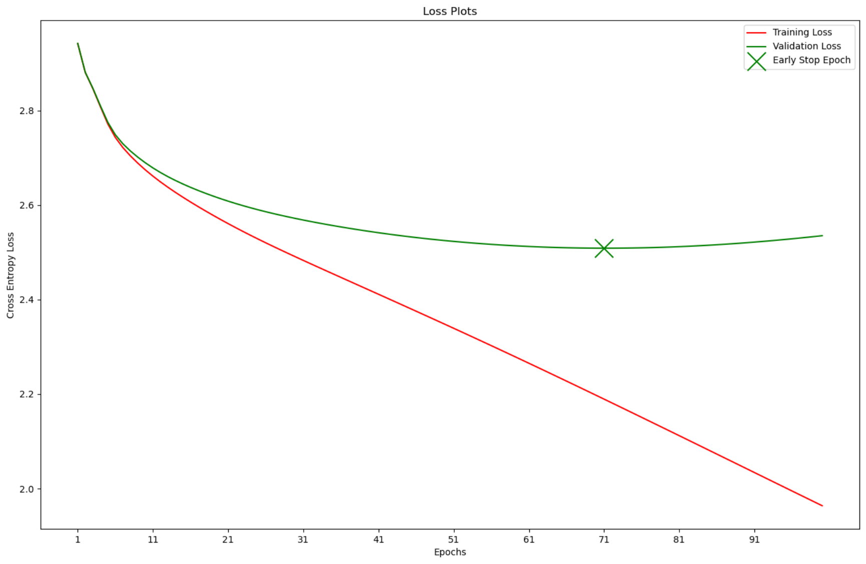

Figure 2: Training and Validation Loss for Momentum Experiment

see convergence even for 150 epochs. Also, the convergence happens quickly for $\alpha = 0.005$, as compared to $\alpha = 0.001$. However, we get a slighty higher accuracy for lower α.

We get the best accuracy (25.92%) for $\alpha = 0.001$ and $\lambda = 0.0001$, with $\lambda = 0.001$ a very close second (25.88%).

Table 4: Hyperparameter tuning for regularization

L_2 **penalty** (λ)	α	**Validation accuracy**	**Early stop epoch**
0.01	0.001	26.29%	-
0.01	0.005	25.71%	39
0.001	0.001	25.88%	74
0.001	0.005	25.42%	15
0.0001	**0.001**	**25.92%**	**71**
0.0001	0.005	25.26%	14

6.2 Results

Figures 4 and 3 represent the loss and accuracy while training for this experiment.

We get a **test set accuracy of 25.37%** using out best hyperparameters ($\lambda = 0.0001$, $\alpha = 0.001$). However, we do get higher accuracies for higher λ. Even though it did not converge for this experiment, we will attempt to use $\lambda = 0.01$ for further experiments, since it seems to be most promising and gives us the highest accuracy of 26.29%.

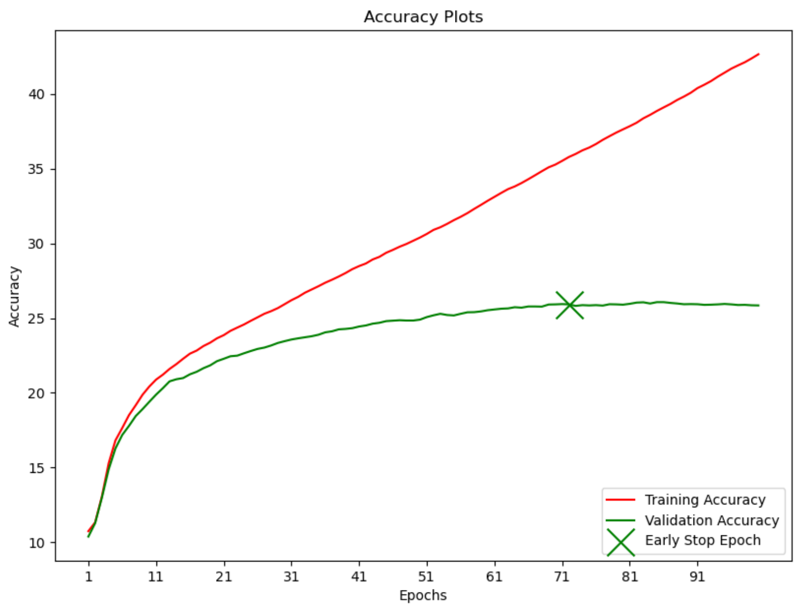

Figure 3: Training and Validation Accuracy for Regularization experiment

7 Activations

Activations are non-linear functions applied to weighted inputs of a hidden layer or the output layer to model complex relationship between the inputs and outputs. For the preceding parts, we have had `tanh` as the activation function for the hidden layer and `softmax` for the output layer.

7.0.1 Hyperparameter tuning

In this part, we have experimented with `ReLU` (Rectified Linear Unit) and `sigmoid` for the hidden layer. While `ReLU` is a widely used activation function for the hidden layers, `sigmoid` is generally

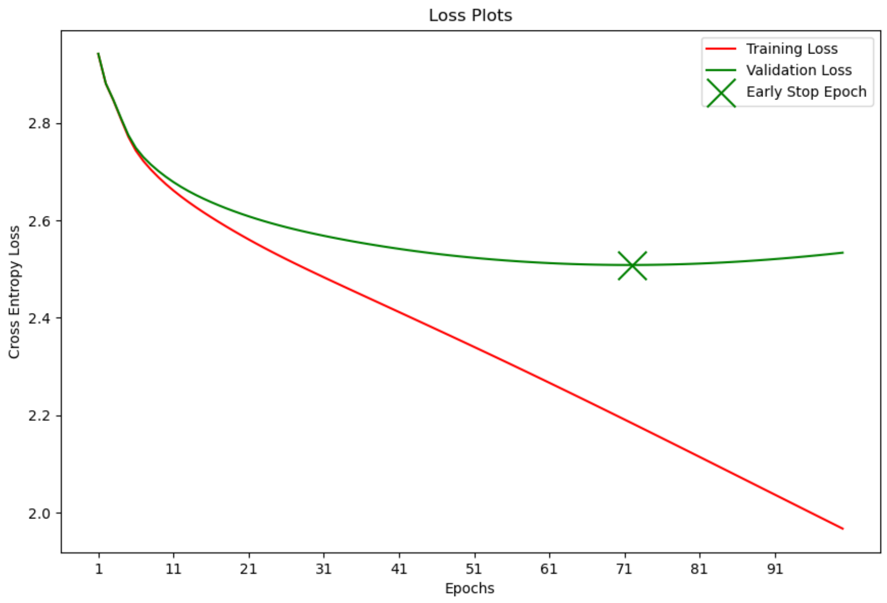

Figure 4: Training and Validation Loss for Regularization experiment

used for output layers of binary classification models as it maps an input value between 0 and 1, similar to probability. `ReLU` is relatively easier to implement as it maps negative values to 0 and positive input values to the same value. Moreover, `ReLU` is devoid of problems relating to vanishing gradients, which is quite prominent with `sigmoid` if implemented in hidden layers of a Neural Network.

While we test our network with the two activation functions for the hidden layer, we also vary the learning rate (α) to get the best possible solution. The results are tabulated in Tables 5. As seen, `sigmoid` was the toughest to converge, as it struggles with the problem of vanishing gradients. There is very little change in the weights; hence, we had to drastically increase the learning rate to 0.05 to induce a convergence point. Although, the result form `sigmoid` function is sub-optimal, it is a good reference point to compare it with `ReLU` and `tanh`.

Table 5: Hyperparameter tuning for activation function

Activation	α	**Accuracy**	**Early stop epoch**
ReLU	**0.001**	**29.69%**	**95**
`ReLU`	0.005	28.76%	18
`sigmoid`	0.005	23.54%	-
`sigmoid`	0.05	21.68%	55

7.1 Results

Based on the best activation function (`ReLU`) and learning rate ($\alpha = 0.001$), accuracy and loss plots for both, training and validation sets are obtained (Fig 5 and Fig 6).

It can be seen from the plots that, the loss is minimized around epoch 95 and the accuracy peaks around that epoch as well. Going by the trend seen form hyperparameter tuning, lowering the learning rate would have probably improved the accuracy; however, at a higher computational cost.

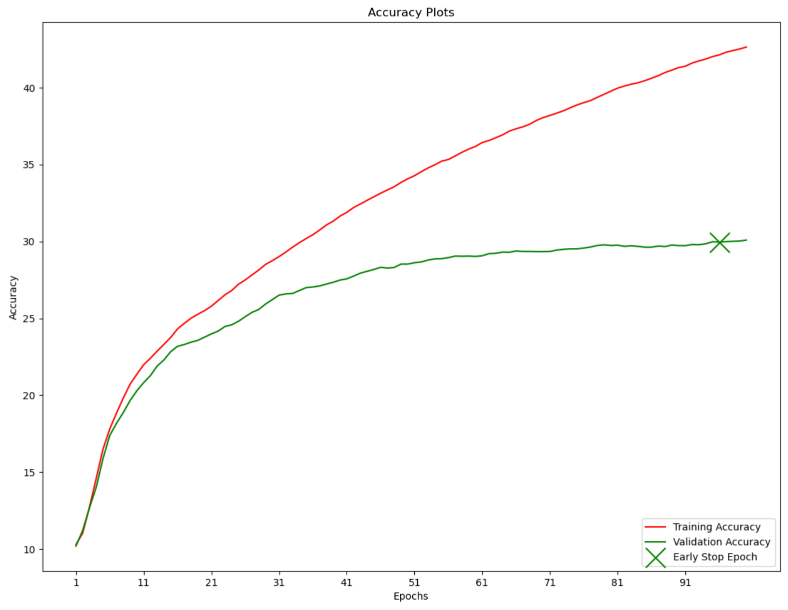

Figure 5: Training and Validation Accuracy for Activation experiment

The test set accuracy at the selected parameters was found to be 29.23%, which is clearly better than what we have predicted in the case of `tanh` activation function.

8 Network topology

Network topology can generally be changed by either

1. Changing the number of hidden layer units
2. Changing the number of hidden layers

Both of these methods end up either increasing/decreasing the number of total parameters that are to be trained.

By increasing the number of hidden layers/units, we increase the number of parameters for the model, thereby its ability to learn more complex data representations and features. However, a higher number of parameters could lead to overfitting of the data as well. Also, deeper networks are more susceptible to the issue of *vanishing gradients*, wherein the gradients become so small that SGD slows down considerably. Thus it is important to have the *right* number of layers, and not necessarily complicate the model more that what is required.

8.1 Change number of hidden units

In this section, we change the number of units in the hidden layer, more precisely, we experiment with halving and doubling the units in the hidden layer, and then quantifying its effects.

8.1.1 Hyperparameter tuning

We observe from Table 6 that for a low α, convergence did not occur.

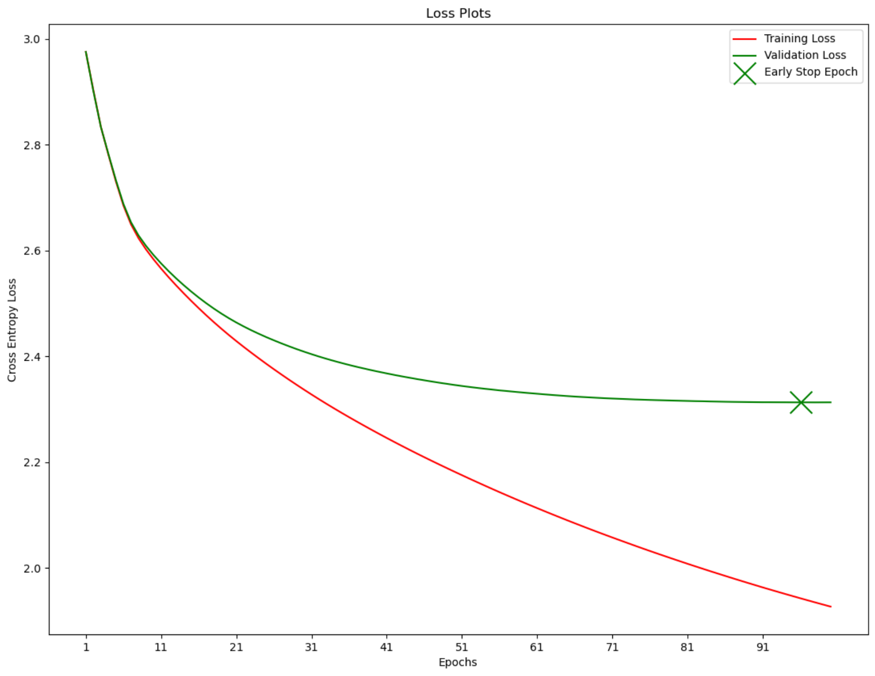

Figure 6: Training and Validation Loss for Activation experiment

Table 6: Hyperparameter tuning for changing number of hidden units

Hidden layer units	α	**Validation accuracy**	**Early stop epoch**
64	0.001	28.14%	-
64	0.003	27.78%	36
64	0.005	27.55%	22
256	0.001	30.38%	-
256	**0.003**	**29.82%**	**33**
256	0.005	29.46%	20

8.1.2 Results

Figures 7 and 8 represent the loss and accuracy while training for this experiment.

Compared to Section 7 (29.69%), we get a validation accuracy of 29.82% for the network with double (256) hidden units. This is expected, since we now capture more features due to more parameters (weights). However, the increase in accuracy is not substantial, and might even be detrimental when training large datasets. At this point, a conscious decision needs to be made regarding the network topology vs computation costs.

The test set accuracy for this experiment was 29.98%.

8.2 Change number of hidden layers

In this section, we change the number of hidden layers and quantifying its effects.

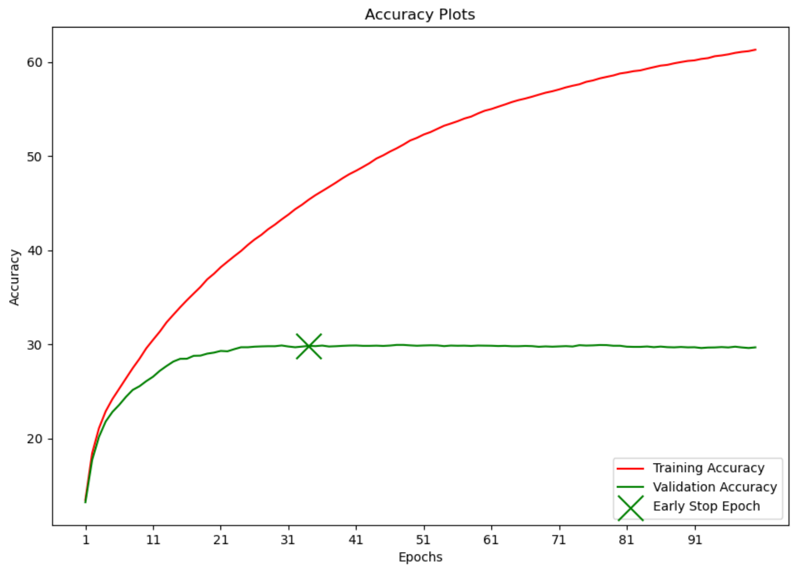

Figure 7: Training and Validation Accuracy for changing hidden units

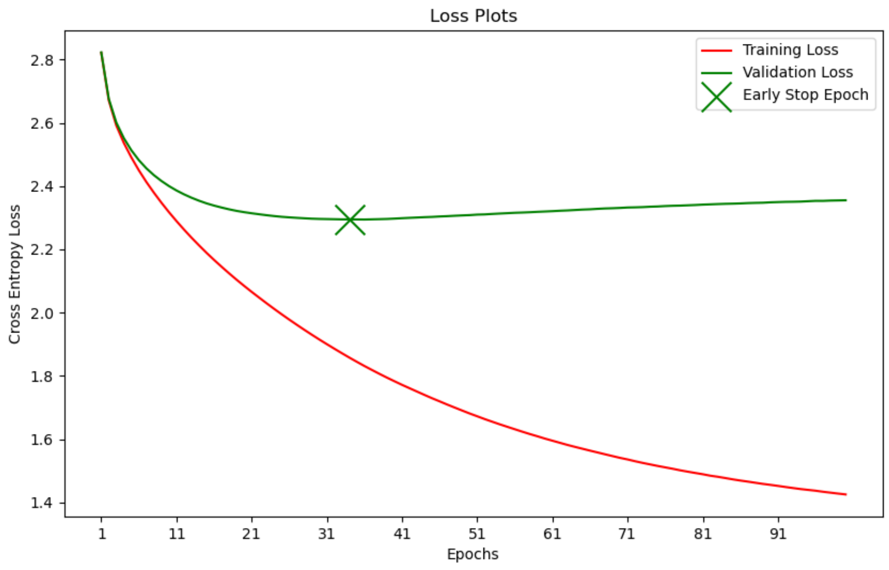

Figure 8: Training and Validation Loss for changing hidden units

8.2.1 Hyperparameter tuning

In order to compare two networks fairly, we change the number of hidden layers/units such that total number of parameters remains the same. From Section 7, we had $(3073 \times 128) + (129 \times 20) = 395,924$ weights. In order to create an additional hidden layer, we would need about 120 units each $(3073 \times 120) + (121 \times 120) + (121 \times 20) = 385,700$. Thus for this experiment, we have used 2 layers of 120 units each for our network.

Table 7: Hyperparameter tuning for changing number of hidden layers

Hidden layers	α	**Validation accuracy**	**Early stop epoch**
2 (120 * 120)	0.001	18.47%	did not converge
2 (120 * 120)	**0.005**	**27.94%**	**50**

We observe from Table 7 that for a low α, convergence did not occur.

8.2.2 Results

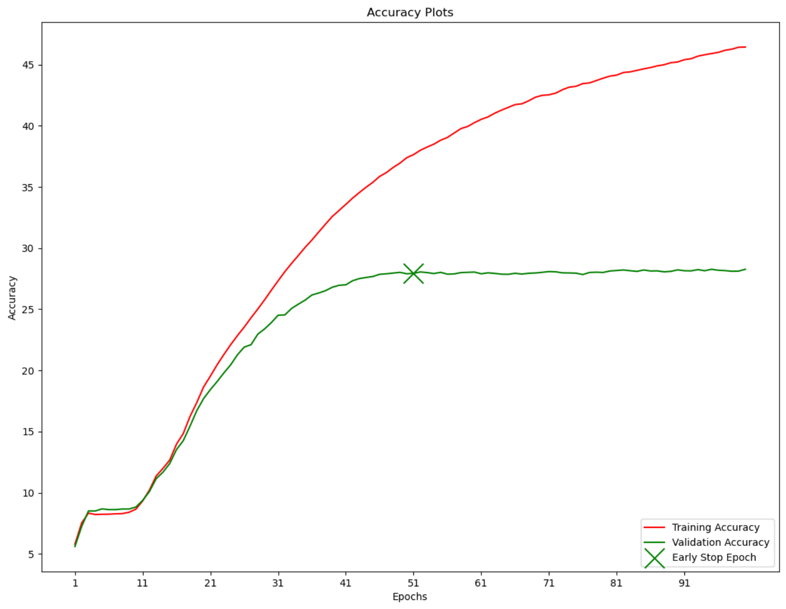

Figure 9: Training and Validation Accuracy for increasing hidden layers

Figures 9 and 10 represent the loss and accuracy while training for this experiment. From Table 7, we see that we obtained best results for $\alpha = 0.005$, with a validation accuracy of 27.94%. This is quite close to the 28.76% accuracy in Table 5. Even though adding in a layer should increase the accuracy, we think it might be due to having $\sim 10,000$ less parameters in the multi-layer model compared to the model from Section 5. Increasing the hidden layer units such that both end up having even more similar number of parameters might lead to the expected result.

The test set accuracy for this experiment was 27.62%.

9 Fine label classification (100 classes)

CIFAR-100 is a dataset consisting of 100 finely labeled classes; however, we have been training and testing our network on 20 coarser classes only. For this experiment, we train our network with all 100 classes and tune the hyperparameters based on validation set results. As we increase, the number of classes by 5 times, it is expected that we should increase the depth and parameters of the neural network as well, but we test our network using the same topology as previous experiments in order to generate comparative results.

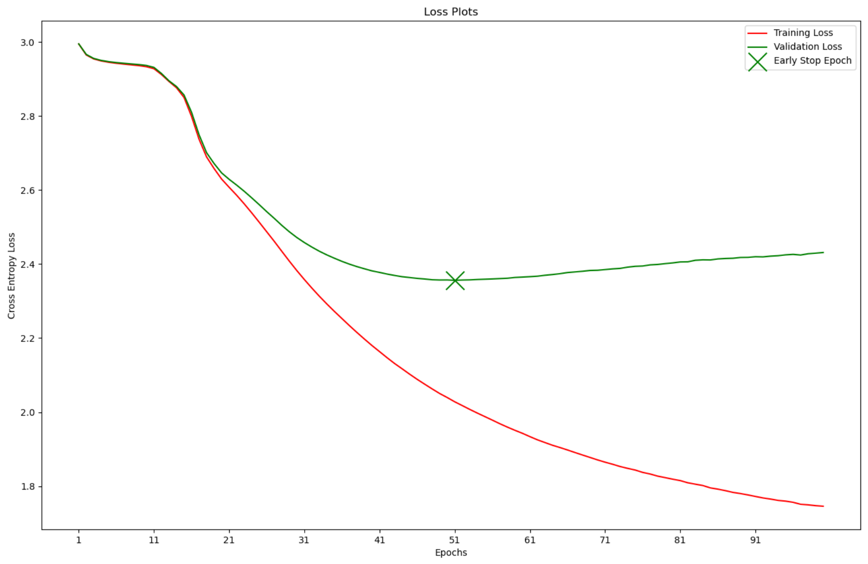

Figure 10: Training and Validation Loss for increasing hidden layers

9.1 Hyperparameter tuning

For 100 class training, it was seen that the gradient descent does not converge with the best parameters used from previous experiments. Hence, we had to lower the regularization penalty $L2$ from 0.01 to 0.001 and tested it on different learning rates (α).

When the regularization penalty is decreased, it allows the model to capture finer details of the training set as larger weights are allowed. With $L2 = 0.01$, the model was seemingly underfitting and was not able to capture the detailed patterns pertaining to 100 classes as compared to 20 classes.

The results on the validation set have been tabulated in Table 8. It can be seen that the accuracy increases with decreasing learning rate; however, it remains far from what has been achieved with 20 classes. Primarily, shallowness of the neural network can be attributed to such discrepancy.

Table 8: Hyperparameter tuning for 100 classes

$L2$	α	**Accuracy**	**Early stop epoch**
0.01	0.001	18.64%	-
0.01	0.005	20.58%	-
0.01	0.010	20.11%	-
0.001	0.001	19.52%	96
0.001	**0.005**	**19.59%**	**19**
0.001	0.010	18.93%	09

9.2 Results

Based on the best learning rate ($\alpha = 0.005$) and regularization penalty ($L2 = 0.001$), accuracy and loss plots for both, training and validation sets are obtained (Fig 11 and Fig 12).

It can be seen from the plots that, the loss is minimized around epoch 19 and the accuracy peaks around that epoch as well. Going by the trend seen form hyperparameter tuning, lowering the

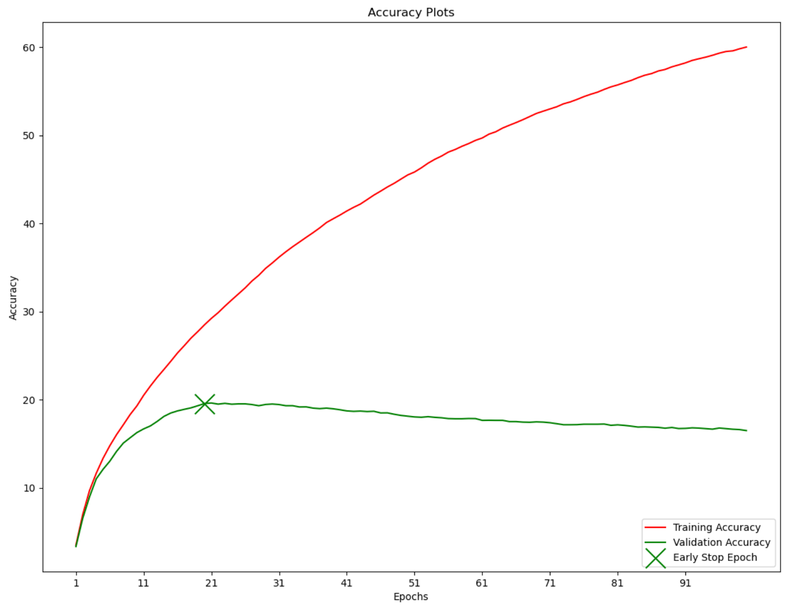

Figure 11: Training and Validation Accuracy for 100 classes

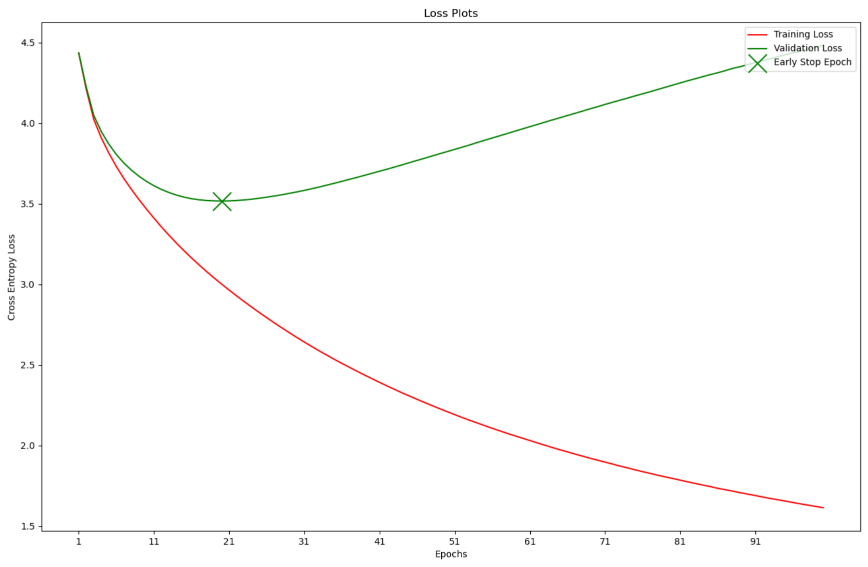

Figure 12: Training and Validation Loss for 100 classes

learning rate would have probably improved the accuracy; however, it decreases as it gets stuck in a local minimum. Hence, we had to choose a bigger α for 100 classes.

The test set accuracy at the selected parameters was found to be 20.13%, which is clearly better than what we have predicted in the case of `tanh` activation function.

Team contributions

- Rishabh:
  - **Code:** `main.py`, `train.py`, `neuralnet.py`, `gradient.py`, `configs.yaml`
  - **Report:** Gradient check, Regularization, Network topology (hidden units and hidden layers)
- Shrey:
  - **Code:** `main.py`, `train.py`, `neuralnet.py`, `gradient.py`, `configs.yaml`
  - **Report:** Momentum, Activations, Fine label classification

For the code, we both worked on developing the backprop algorithm, testing the hyperparameters for different configs, and filling out the auxiliary files (`gradient.py` and `train.py`).